Increasing Generality in Machine Learning through Procedural Content Generation

Sebastian Risi[1,2] and Julian Togelius[1,3]
[1]modl.ai, [2]IT University of Copenhagen, [3]New York University
{sebastian, julian}@modl.ai

Procedural Content Generation (PCG) refers to the practice, in videogames and other games, of generating content such as levels, quests, or characters algorithmically. Motivated by the need to make games replayable, as well as to reduce authoring burden, limit storage space requirements, and enable particular aesthetics, a large number of PCG methods have been devised by game developers. Additionally, researchers have explored adapting methods from machine learning, optimization, and constraint solving to PCG problems. Games have been widely used in AI research since the inception of the field, and in recent years have been used to develop and benchmark new machine learning algorithms. Through this practice, it has become more apparent that these algorithms are susceptible to overfitting. Often, an algorithm will not learn a general policy, but instead a policy that will only work for a particular version of a particular task with particular initial parameters. In response, researchers have begun exploring randomization of problem parameters to counteract such overfitting and to allow trained policies to more easily transfer from one environment to another, such as from a simulated robot to a robot in the real world. Here we review the large amount of existing work on PCG, which we believe has an important role to play in increasing the generality of machine learning methods. The main goal here is to present RL/AI with new tools from the PCG toolbox, and its secondary goal is to explain to game developers and researchers a way in which their work is relevant to AI research.

1 Introduction

For several decades, procedural content generation (PCG) has been a feature of many video games (Table. 1). PCG refers to the algorithmic creation of game content — not the game engine, but things such as levels, quests, maps, characters or even rules — either in runtime (as the game is being played) or at design time (as the game is made). There are several reasons why PCG is used in games: it can increase the replayability of a game as players are presented with a new experience every time they play, it can help to reduce production costs and disk storage space, and it enables new types of games built on the unique affordances of content generation.

Interestingly, developments in PCG and machine learning have started to influence each other in reciprocal ways. *Procedural Content Generation via Machine Learning* (PCGML) [1] refers to the use of machine learning to train models on existing game content, and then leverage these models to create novel content automatically. This can be done through simply sampling from the learned models, or through searching the artifact space implied by the model so as to optimize some objective. In this paper, the term artifact refers to objects/levels/maps/etc. made by an algorithm. Interestingly, PCGML poses different and hard challenges compared to generating e.g. images, because the produced content needs to function.

At the same time as PCG researchers are starting to incorporate these advances into their systems, interest in the machine learning community is increasing in PCG-inspired methods to improve the robustness of ML systems. One reason for this development is the growing evidence that, while ML methods perform well for tasks or in the environments they are trained on, they do not generalize well when that environment is changed or different from what is seen during training. Training neural networks with many free parameters and over long training times has lead to state-of-the-art performance in many domains, but these solutions are typically overfitted to the particular training examples, achieving high accuracy on a training set but performing poorly on data not used for training [2, 3]. Especially in deep reinforcement learning (in which an agent has to learn in interaction with its environment), overfitting is rarely addressed [4, 5, 6] but a significant problem. Take the very popular Arcade Learning Environment (ALE) as an example [7], a classic benchmark in RL based on an emulation of the Atari 2600 games console. Hundreds of games were made for that console, however they all have fixed sets of levels and very little in the way of randomization. Training an agent to play a game in ALE makes it liable to overfit not only to that particular game, but to its levels and sequence of events.

The basic idea in employing PCG to address the generality problem in ML systems is to artificially create more training data or training situations. This way ML-based systems can be biased towards learning general task properties instead of learning spurious elements found in the training examples. Methods include simpler approaches such as data augmentation that artificially increase the data used for training [8, 9] or methods that train agents in a large number of training environments that include randomized elements [10]. PCG methods have been extended to create complete maps for Cap-

ture the Flag [11] or maps for 2D video games [12], and in a recent impressive demonstration of the advantage of training in PCG-generated environments, have allowed a robot hand trained in a simulation to manipulate a Rubik's cube in the real world [13].

In this article, we review the history of PCG and the recent trends in hybridizing PCG methods with machine learning techniques. We do this because we believe there are convergent research interests and complementary methods in the two communities. We want to supply machine learning researchers with a new toolbox to aid their generalization work, and games researchers and developers with new perspectives from machine learning. This Review also details promising future research direction and under-explored research avenues enabled by more advanced PCG techniques. The goal of this Review is to share with the larger machine learning research community work from the exciting field of PCG that is just beginning to capture the interest of ML researchers but could ultimately encourage the emergence of more general AI. We focus on examples where there's a notion of "environment" (typically through learning being centered on an agent in simulated physical world), but will throughout the text mention other learning settings where relevant for comparison.

We will discuss work coming out of both commercial game development, AI/ML research targeted at games, and AI/ML research targeted at other applications or the development of general intelligence. It is important to bear in mind that these approaches were developed for very different purposes, with those coming out of game development and research typically focused on providing entertainment. However, this should not in general make them less suitable for other purposes. Indeed, as one of the reasons games are entertaining is that they challenge our minds, one could argue that algorithms designed to increase entertainment in games may be useful for creating intelligence-relevant challenges [14].

We will discuss both work that we classify as PCG and certain examples of work that is adjacent to PCG, such as some forms of data augmentation in supervised learning. Our necessarily imperfect line of demarcation is that pure randomization/shuffling is not PCG; however, many PCG algorithms include randomness.

2 Classic Procedural Content Generation

While possibly the first video game to include PCG dates from 1978 (Beneath Apple Manor by Don Worth for the Apple II), Rogue (1980) by Toy and Wichmann created an important design paradigm. In Rogue (Fig. 1a), the player explores a multi-level dungeon complex, battling enemies and collecting treasures. As the creators did not want to author the dungeons themselves (they wanted to play the game and be surprised), they needed to create a dungeon generation algorithm; every time you play a game of Rogue, a new set of dungeons are generated. Rogue came to inspire a genre of games called *rogeuelikes*, which are characterized mainly by the use of runtime generation of content that is essential to gameplay. The highly successful Diablo series of games (Blizzard, 1997-2013) (Fig. 1c), as well as platformers such as *Spelunky* (Mossmouth, 2008), are roguelikes.

While the PCG in Rogue was motivated by a need for replayability and unpredictability, another key reason for PCG is wanting to create game worlds larger than can fit in memory or on storage media. A paradigm-setting game here was *Elite* (Brabensoft, 1984), a space-faring adventure game featuring thousands of planets which seemingly miraculously fit in memory on a Commodore 64, with 64 kilobytes of memory (Fig. 1b). Every time a star system was visited, the game would recreate the whole starsystem with planets, space stations, and spacecraft, from a given random seed. This approach has later been used for games such as *No Man's Sky* (Hello Games, 2015), which famously contains more planets than you can visit in a lifetime, all with their own ecologies (Fig. 1e).

The strategy games in the very popular *Civilization* series also rely heavily on PCG, as a new world is created for the players to explore and contest every time a new game is created (Fig. 1f). Similarly, the open-world sandbox game *Minecraft* (Mojang, 2010) creates a completely new world at the start of each game session (Fig. 1d). Other games use PCG in somewhat more peripheral roles, such as the sidequest generation (e.g. creating an infinite supply of fetch quests through a guild system) in *The Elder Scrolls V: Skyrim* (Bethesda, 2011) (along with some earlier games in the series) and the pervasive generation of terrain features and vegetation in a large number of open-world 3D games. PCG techniques are now so commonplace and reliable that it is more common than not to utilize them in many game genres.

Interestingly, PCG in video games is actually prefigured by certain pen-and-paper generators intended to be executed by humans with the help of dice or cards, including a dungeon generator for the classic *Dungeons and Dragons* (TSR, 1976) role playing game [31]. Some recent board games which include aspects of PCG are *504* (2F Spiele, 2015) or *Betrayal at House on the Hill* (Avalon Hill, 2004).

The types of PCG that can be found in most existing games are called *constructive* PCG methods (Table 1). This means that the content generation algorithm runs in a fixed time, without iteration, and does not perform any search. For generating textures, heightmaps, and similar content, a commonly used family of algorithms are fractal noise algorithms such as Perlin noise [32]. Vegetation, cave systems, and similar branching structures can be efficiently generated with graphically interpreted grammars such as L-systems [33]. Other constructive methods that are borrowed from other fields of computer science and adapted to the needs of PCG in games include cellular automata [34] and other approaches based on local computation. Other constructive methods are based on rather less principled and more game-specific methods. For example, *Spelunky* combines a number of pre-authored level chunks according to patterns which are designed so as to ensure unbroken paths from entrance to exit.

Table 1: A comparison of several methods for PCG and domain randomization described in this article. The first two columns indicate the representation of the content, which is either designed by hand or learned through machine learning. The last columns indicate how the content is generated given a representation. Constructive methods follow rules and do not do any resampling. Dwarf Fortress is an example of a generate-and-test (random search) method where the world is regenerated if it fails certain tests. Most PCGML approaches randomly sample a learned representation, whereas PCGML with constraints resample when constraints are not satisfied. In the search-based paradigm, a hand-coded representation is searched using an evolutionary algorithm. Latent Variable Evolution combines search-based PCG with a learned representation, e.g. in the form of a GAN. PCGRL uses a policy learned by reinforcement learning to search a hand-coded representation, whereas Generative Playing Networks instead uses reinforcement learning to test the artifacts and gradient descent to generate them. Activation Maximization relies on gradient descent to generate artifacts, based on a learned representation. The various forms of domain randomization use hand-coded representations and differ in whether they simply sample this space or perform some sort of search with a learned policy. POET and MCC are fundamentally search-based methods, which include a learning agent inside the evaluation loop. Progressive PCG uses a parameterizable constructive generator, coupled to a RL-based game-playing agent.

	Representation		Generation Method					
	Learned	Hand-designed	Evol.	Learned	Gradient-based	Rand. search	Sampl.	Rules
Standard Constructive E.g. Rogue, Pitfall!, Civilization, Elite, Minecraft	○	●	○	○	○	○	○	●
Dwarf Fortress	○	●	○	○	○	●	○	○
Standard PCGML [1, 15, 16]	●	○	○	○	○	○	●	○
PCGML with constrained sampling [17]	●	○	○	○	○	●	○	○
Standard search-based [18, 19, 20, 21, 22]	○	●	●	○	○	○	○	○
Latent Variable Evolution [23, 24]	●	○	●	○	○	○	○	○
PCGRL [25]	○	●	○	●	○	○	○	○
Generative Playing Networks [26]	●	○	○	○	●	○	○	○
Activation Maximization [27]	●	○	○	○	●	○	○	○
Simple Data Augmentation [8, 9]	●	○	○	○	○	○	○	●
Uniform Domain Randomization [10]	○	●	○	○	○	○	●	○
Guided Domain Randomization [28]	○	●	○	●	○	○	○	○
Automatic Domain Randomization [13]	○	●	○	○	○	○	●	○
POET [29], MCC [30]	○	●	●	○	○	○	○	○
Progressive PCG (PPCG) [4]	○	●	○	○	○	○	○	●

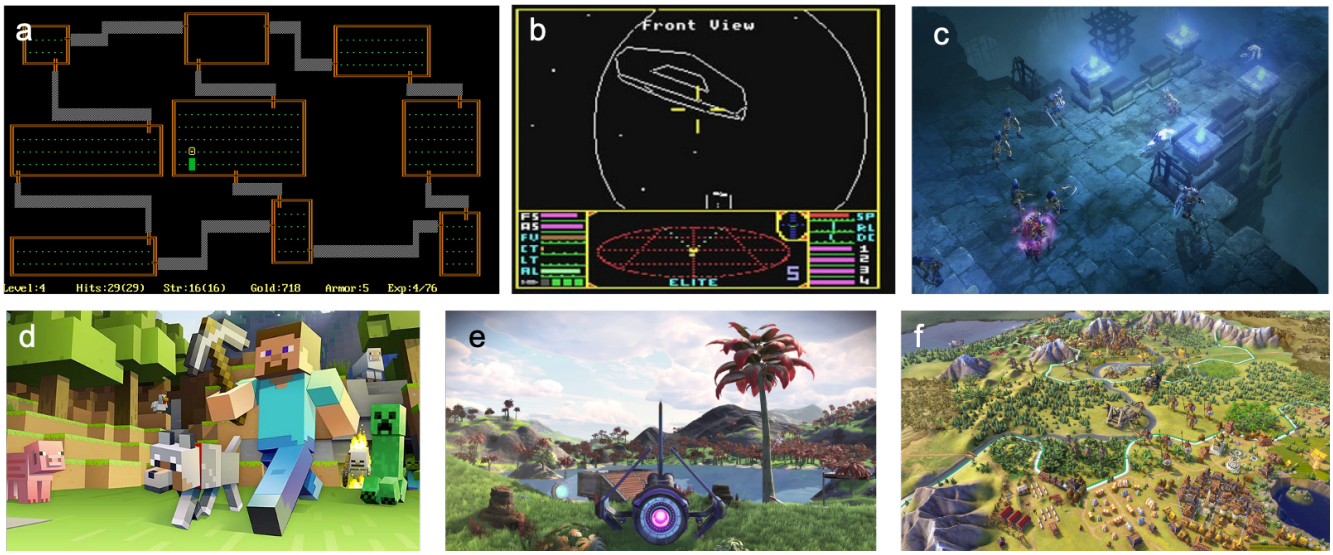

Figure 1: Example commercial games that feature PCG as an important game component are Rogue (**a**), Elite (**b**), Diablo III (**c**), Minecraft (**d**), No Man's Sky (**e**), and Civilisation VI (**f**).

3 "PCG" in Machine Learning: Data Augmentation and Domain Randomization

While not necessarily called PCG in the machine learning community, the idea of *data augmentation* is essentially a simple form of constructive PCG. The aim of data augmentation is to increase the diversity in the dataset, not by collecting more data but by adding modified versions of the already existing data [8, 9]. Data augmentation is very common in supervised learning tasks, for example, through cropping, padding, or adding noise to images in a dataset. It is common practice in machine learning and has resulted in significantly less overfitting and state-of-art results in a variety of domains [8, 35, 36].

A different form of data augmentation was introduced by Geirhos et al. [37], in which the authors showed that training the *same* network architecture but with a stylized version of ImageNet images (e.g. a cat with the texture of an elephant) can significantly increase the model's accuracy and robustness. In fact, the authors showed that a deep convolutional network trained on the standard ImageNet mainly focuses on textures in images instead of their shape; training on the stylized version of ImageNet increases their shape bias and with that, their accuracy and robustness.

In the field of reinforcement learning, domain randomization [38, 10, 39] is a simple form of PCG and one way to counter overfitting in machine learning. The main idea of domain randomization is to train an agent in many simulated training environments, where certain properties are different in each environment. The goal is to learn a single policy that can work well across all of them. In addition to trying to encourage machine learning systems to be more robust and general, another use case of domain randomization is to facilitate the transfer of policies trained in a simulator to the real world [10, 40, 41, 42]. Training in a simulation instead of the real world has several advantages such as the training being faster, cheaper and more scalable, and having access to the ground truth.

In a promising demonstration of this approach, Tobin et al. [10] trained an object detector on thousand of examples of objects with randomized colors, textures, camera positions, lighting conditions, etc. in a simulator and then showed it can detect objects in the real world without any additional training. Another example is the work by Sadeghi et al. [40], who trained a vision-based navigation policy for a quadrotor entirely in a simulated environment with highly randomized rendering settings and then transferred this policy to the real world without further training.

Following Weng [38], we can further divide domain randomization into three subgroups: *uniform domain randomization*, *guided domain randomization*, and *automatic domain randomization*. In uniform domain randomization, each parameter is uniformly sampled within a certain range. For example, in the work by Tobin et al. [10], the size of objects, their mass, or the amount of noise added to the camera image were drawn from a uniform distribution.

Guided domain randomization. In the more sophisticated guided domain randomization, the type of randomization is influenced by its effect on the training process [43, 28, 38, 44]. The goal of this guided randomization is to save computational resources by focusing the training on aspects of the task that actually increase the generality of the model. For example, instead of randomly applying pre-defined and hard-coded data augmentation methods, the approach AutoAugment [28] can learn new data augmentation techniques. These augmentation techniques are optimized for based on their validation accuracy on the target dataset. Such methods can be seen as a form of adaptive content generation; in the PCG literature there are approaches to PCG that adapt to an agent-driven by e.g. Schmidhuber's theory of curiosity [45, 46].

Another related approach is DeceptionNet [44], which is trained to find modifications to an image through distortion, changing the background, etc. that make it harder for an image recognition network to output the correct classification. Both a recognition and deception network are alternatively trained such that the deception module becomes better in confusing the recognition module, and the recognition module becomes better in dealing with the modified images created by the deception module.

Automatic Domain Randomization (ADR). Very recently, OpenAI showed that a neural network that controls a five-fingered humanoid robot hand to manipulate a Rubik's cube, can sometimes solve this task in the real world even though it was only trained in a simulated environment [13]. Key to this achievement in robotic manipulation was training the robot in simulation on a large variety of different environmental variations, similar to the domain randomization approaches mentioned above. Following related work in PCG for games [4], the ingredient to make this system work was to increase the amount of domain randomization, as the robot gets better and better at the task. For example, while the network was initially only tasked to control a Rubik's cube of 5.7 cm, later in training it had to deal with cubes that could range from 5.47 to 6.13 cm in simulation. Because the robot had to deal with many different environments, dynamics of meta-learning did emerge in the trained neural network; this allowed the robot to adapt to different situations during test time, such as the transfer to the real world. ADR is similar to guided domain randomization but focuses more on increasing the diversity of the training environments based on task performance, instead of sampling efficiently from a distribution of environments.

While current domain randomization methods are showing promising results, the PCG community has invented many sophisticated algorithms that – we believe – could greatly improve the generality of machine learning methods even further. As we discuss in the following sections, more recent work in PCG has focused on search-based approaches and on learning the underlying PCG representations through machine learning techniques.

4 AI-driven Procedural Content Generation

Given the successes of PCG in existing video games, as well as the perceived limitations of current PCG methods, the last decade has seen a new research field form around game content generation. The motivations include being able to generate types of game content that cannot currently be reliably generated, making game developments easier and less resource-intensive, enabling player-adaptive games that create content in response to player actions or preferences, and generating complete games from scratch. Typically, the motivations center on games and players, however, as we shall see, many of the same methods can be used for creating and varying environments for developing and testing AI.

While there has been recent work on constructive methods, more work has focused on approaches based on search and/or machine learning.

4.1 Search-based PCG

In search-based PCG (Table 1), stochastic search/optimization algorithms are used to search for good content according to some evaluation function [18]. Often, but not always, some type of evolutionary algorithms is used, due to the versatility of these algorithms. Designing a successful search-based content generation solution hinges on designing a good representation, which enables game content to be searched for. The representation affects, among other things, which algorithms can be used in the search process; if the content can be represented as a vector of real numbers, this allows for very strong algorithms such as CMA-ES [47] and differential evolution [48] to be used. If the representation is e.g. a graph or a permutation, this poses more constraints on the search.

An early success for search-based PCG is Browne and Maire's work on generating board games, using a game description language capable of describing rules and boards for classical board games [20]. The initial population was seeded with a dozens of such games, including *Checkers*, *Connect Four*, *Gomoku*, and *Hex*. The evaluation function was simulation-based; candidate games were evaluated through being played with a Minimax algorithm combined with a state evaluation function automatically derived for each game. The actual game evaluation is a combination of many metrics, including how often the game leads to a draw, how early in the game it is possible to predict the winner, and number of lead changes. This process, though computationally very expensive, came up with at least one game (*Yavalath*), which was of sufficient quality to be sold commercially (Fig. 2f).

While attempts at creating complete video games including game rules through search based-methods have met mixed success [46, 52, 49, 53], search-based PCG has been more effective in generating specific types of game content such as levels. We have seen applications to generating maps for the real-time strategy game StarCraft [21], and levels for the platform game Super Mario Bros [19], the first-person shooter Doom [54], and the physics puzzle game Angry Birds [55], among many similar applications. Search-based PCG has also been used for other types of game artifacts, such as particle effects for weapons [56], role-playing game classes [57] and flowers [51] (Fig. 2e).

The most important component in a search-based content generation pipeline is the evaluation function, which assigns a number (or vector) to how desirable the artifact is. In many cases, this is accomplished through playing through the content in some way and assigning a value based on characteristics of the gameplay, as in the Ludi example above; other evaluation functions can be based on directly observing the artifact, or on some machine-learned estimate of e.g. player experience.

An emerging trend is to go beyond optimizing for a single objective, and instead trying to generate a diverse

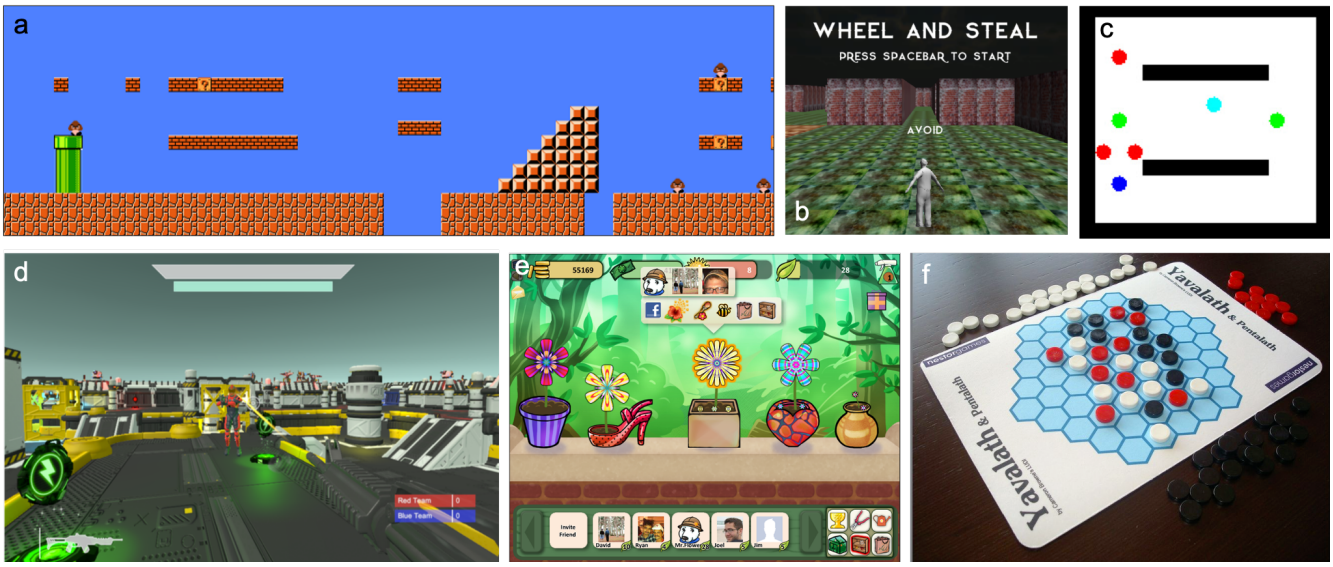

Figure 2: In academia, PCG approaches have been used to produce complete and playable (**a**) Super Mario Bros. levels [16], (**b**) 3D [49], (**c**) 2D games [46], and (**d**) maps and character classes for first-person shooters [50]. Other PCG-enabled games include Petalz (**e**), in which players can collaboratively breed an unlimited variety of different procedurally generated flowers [51]. The game Yavalath (**f**) is one of the few examples of PCG-generated games that are commercially available.

set of artifacts that perform well. The goal here is to generate, for example, not just a single level but a set of levels that vary along various dimensions, such as number of enemies of difficulty to solve for an A* algorithm [58]. The Map-Elites algorithm in particular, originally introduced to create more robust robot gaits [59], has been adapted to create sets of game levels that vary in what skills they require from the agent or what mechanics they feature [60].

An alternative to stochastic optimization algorithms is to use constraint satisfaction methods [61]) such as answer set programming [62]. Casting artifacts as answer sets can allow very efficient search for content that obeys specific constraints, but is hard to integrate with simulation-based evaluation methods. This paradigm is sometimes called *solver-based* PCG.

4.2 PCG via machine learning

Machine learning methods such as Generative Adversarial Networks (GANs) [63] have revolutionized the way we generate pictorial content, such as images of faces, letters, and various objects. However, when generating game content with some form of playability constraints (such as levels, maps, or quests), things become more complicated because these types of content are in some ways more like code than like images. An image of a face where the contours smudge just looks slightly off, whereas a level for Super Mario Bros with an impossibly long jump is not just somewhat defect, it's unplayable and therefore worthless. Similar functionality requirements can be found in level-like artifacts such as robot path planning problems, logic puzzles, and quests [1]. Therefore we call such content, in which some algorithmic way of verifying their functionality (e.g. playability) exists, *functional content*.

Simply training a GAN on a large set of functional artifacts does not guarantee that the generator network learns to produce levels that fulfill these functionality requirements, nor that the discriminator learns to identify and check for those constraints. The result is often simply artifacts that look right but don't function well [12]. Another potential reason for the failure of machine learning-based methods to generate functional content is that methods such as GANs mostly learn local dependencies, whereas functionality in many types of content can depend on features that are far from each other, and/or counting the number of instances of a feature.

The same effect has been found with other representations, such as LSTM networks [15] and Markov chains [17]. One way of counteracting this effect is bootstrapping, where newly generated artifacts that are found to satisfy the functionality requirements are added back to the training set for continued training, thus biasing training specifically to functional artifacts [12].

Machine learning models can also be combined with search to improve their efficiency. One way to do this is to use the learned model as a representation for search-based PCG. The idea here is to use machine learning to find the general space of content which is roughly defined by the examples the model is trained on, and then search within that space. Using GANs, this could be done by searching the *latent space*; when training a GAN, a latent vector is used as input to the generator network. The latent space is defined by that input. *Latent Variable Evolution* refers to using evolutionary algorithms to search the latent space for artifacts that optimize some kind of objective function [24]. For example, latent variable evolution was used to generate new levels for Super Mario Bros, by first training a GAN on one-screen segments of

most levels in the original game. The latent space was then searched for vectors that would maximize objectives such as that the segment should contain many jumps, or should not be winnable without jumping, or should be unwinnable [23].

Functionality evaluation can be integrated into adversarial learning processes in other ways. Generative Playing Networks consist of a generator network, that generates levels, and a reinforcement learning agent that learns to play them [26]. While the objective for playing agent is simply to perform as well as possible on the level, the objective for the playing agent is to provide an appropriate level of challenge for the agent.

Another way of using machine learning for PCG is to use reinforcement learning. The conceptual shift here is to see PCG as a sequential process, where each action modifies a content artifact in some way. The goal of the training process then becomes to find a policy that for any content state selects the next action so that it leads to maximum expected final content quality. For this training process to be useful, we will need the trained policy to be a content generator capable of producing diverse content, rather than simply producing the same artifact every time it is run. A recent paper articulates a framework for PCG via reinforcement learning and proposes methods for ensuring that the policy has sufficiently diverse results in the context of generating two-dimensional levels [25]. Two important lessons learned is to always start from a randomized initial state (which need not be a functional level) and to use short episodes, to prevent the policy from always converging on the same final level. (It is interesting to note that the issues with learning general policies in RL recur in trying to learn policies that create content that can help generalize RL policies.)

Compared to PCG based on supervised or self-supervised learning, PCG based on reinforcement learning has the clear advantage of not requiring prior content to train on, but the drawback of requiring a reward function judging the quality of content. This is very similar in nature to the evaluation function in search-based PCG. Compared to search-based PCG, PCG via reinforcement learning moves the time and computation expense from inference to training stage; whereas search-based PCG uses extensive computation in generating content, PCG via reinforcement learning uses extensive computation to train a model which can then be used cheaply to produce additional content.

5 Procedurally generated learning environments

An exciting opportunity of PCG algorithms is to create the actual learning environments that scaffold the learning of artificial agents (Fig. 3). Similarly to how current machine learning methods are moving towards automating more and more facets of training (e.g. meta-learning the learning algorithms themselves, learning network architectures instead of hand-designing them), the automated generation of these progressive curricula that can guide learning offers unique benefits.

One of the first examples of this idea is *Minimal Criterion Coevolution* (MCC) [30]. In MCC both the agent and the environment co-evolve to solve increasingly more difficult mazes. Recent work building on these ideas is POET [29], which deals with the more challenging OpenAI gym bipedal walker domain. POET is a good example of an approach in which solutions to a particular obstacle-course can function as stepping stones for solving another one. In fact, for the most difficult environments (shown on the right in Fig. 3a) it was not possible to directly train a solution; the stepping stones found in other environments were necessary to solve the most ambitious course.

Importantly, procedurally generated training environments can also increase the generality of reinforcement learning agents. Zhang et al. [5] showed that training on thousands of levels in a simple video game can allow agents to generalize to levels not seen before. Some domains in the OpenAI gym training environments include procedurally generated content, requiring the agents to learn more general strategies. For example, in the CarRacing-v0 environment [64], agents are presented with a new procedurally generated car racing track every episode and the final reward is the average reward over multiple rollouts. These procedurally generated environments required more sophisticated neural architectures to be solvable [65], highlighting their usefulness in testing the ability of the reinforcement learning agents. A similar approach for encouraging the discovery of general policies worked well for evolving stable policies for a 2D bipedal walker domain [66]. In addition to helping in supervised learning settings (see Section 3), forms of data augmentation can also help RL agents to become more robust. Randomized environments are also present in the Arena multi-agent testbed [67]. In the work by Cobbe et al. [68] agents are trained in environments in which random rectangular regions of the environment are cut out and replaced by rectangles filled with random colors, which helps these agents to generalize better.

Jaderberg et al. [11] relied on a PCG-based approach to allow RL agents to master Quake III Arena Capture the Flag (Fig. 3b). In their work, agents were trained on a mixture of procedurally generated indoor and outdoor maps (with varying walls and flag locations), which allowed the agents to learn policies that are robust to variations in the maps or the number of players. This work also demonstrated another advantage of procedurally generated maps: because each map is different, agents learned to learn how to keep track of particular map locations (e.g. the entrance to the two bases) through their external memory system.

While the aforementioned work [5, 11] showed that training on a larger variety of environments can lead to more general agents, it did require a large number of training levels. In work from the PCG research community, Justesen et al. [4] introduced a *Progressive PCG* approach (PPCG), which showed that performance of training agents can be increased while using less data if the difficulty of the level is changed in response to the performance of the agents (Fig. 3d). A similar approach was then later adopted by OpenAI to train their

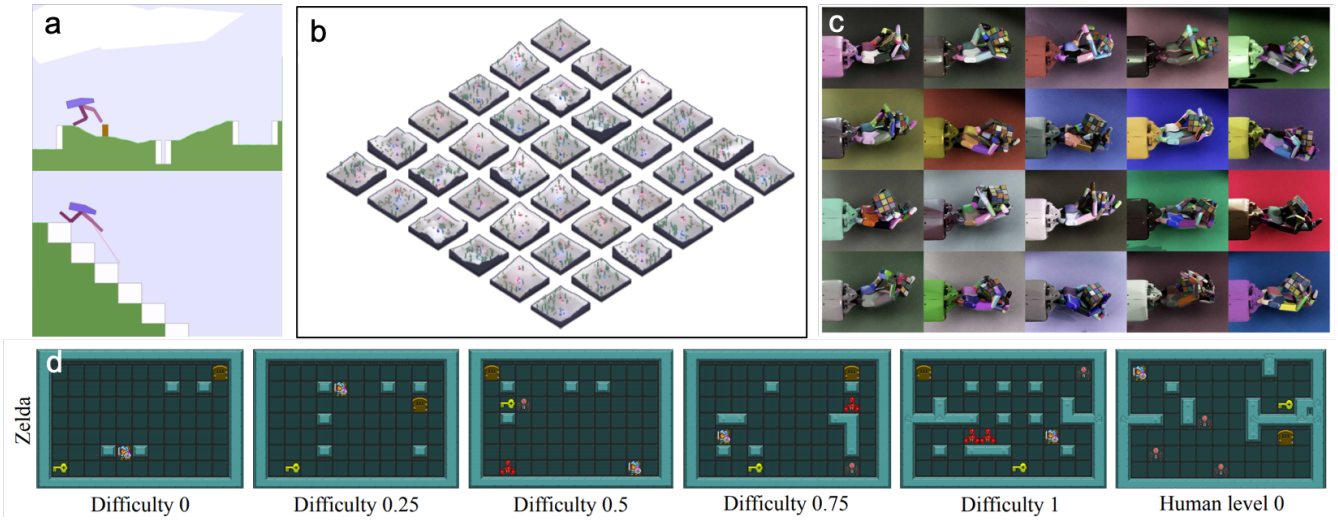

Figure 3: Examples of learning environments created by PCG-based approaches. The POET algorithm (**a**) learns to create increasingly complex environments for a 2D bipedal walker together with their neural network controllers [29]. (**b**) Procedurally generated maps were one of the key ingredients to allow agents to master the Quake III Capture the Flag domain. Increasing task complexity depending on the performance of the agent has shown to lead to more general solutions for (**c**) controlling a robot hand to manipulate a Rubik's cube in simulation and in the real world [13], and (**d**) video game playing [4].

humanoid robot hand (Fig. 3c) in increasingly more challenging environments [13].

In fact, there is some initial evidence that very varied training environments can also foster the emergence of meta-learning in recurrent neural networks, that allow adaption to situations not seen during training [13]. While approaches such as OpenAI's Rubik's cube solving robot hand hint at the potential of this approach, creating an encoding that can produce an even larger variety of different and effective training environments could have a significant impact on the generality of the agents and robots we are able to train.

We also summarise the similarities and differences between POET, MCC, and PPCG in Table 1. While all three approaches use hand-designed representations, PPCG does not evolve the levels but instead uses a ruled-based generator.

6 Looking forward: Opportunities and Challenges

With more advanced ML techniques, PCG approaches are becoming better and better at generating content while PCG methods are now starting to allow more general machine learning systems to be trained. We believe the idea of automatically and procedurally generating learning environments with the right complexity that scaffold the learning of autonomous agents is an exciting research direction that can help overcome some of the constraints that impede generalization and open-ended learning in current AI. This research direction is similar to what has been proposed in PCG research before, and also to the idea of AI-generating algorithms (AI-GA) [69]. We identify five main open challenges that we believe are essential in pushing the field of PCG forward and to realizing its promise to create more adaptive and lifelong learning ML agents.

6.1 Learning from limited data

When generating images with machine learning, it is common practice to train the model on thousands, maybe even millions, of images [70]. However, such amounts of high-quality data are rarely available when developing a game, or even in a finished game. For example, the original *Super Mario Bros* game has 32 levels, resulting in a few hundred screens' worth of content. For some games, a large amount of user-generated content is available online, but this content can be of very variable quality. And when creating, for example, a robot learning benchmark from scratch, creating scenarios to train a content model on can be a significant time investment. Bootstrapping in PCG [12] (see Section 4.2) can help overcome this problem of content shortage, and various data augmentation could also help but learning to generate new content from limited data is still a significant challenge. More research is needed on how to learn from little data, and on how to learn generative models based on many different types of data. For example, training a model on lots of available benchmark rules to learn generic patterns, it should be possible to generate environments for a new benchmark.

6.2 Generating Complete Games

While PCG techniques have shown impressive results for particular types of content in particular game genres, there has been much less progress on the harder problem of generating complete games. Browne and Maire's work from 2010 (discussed above [20]), which resulted

in a well-reviewed board game which is sold in stores, remains the gold standard. Generating complete video games [46, 53, 62, 71, 52, 49] (Fig. 2) or card games [72] seems to be much harder challenges, with the results often being unplayable or uninteresting. Methods that have been tried include constraint satisfaction through answer set programming as well as evolutionary search. This is partly because of these games being very complex, and partly because it is very hard to find good evaluation metrics for complete games. Yet, generating complete challenges, including rules, topology, visuals etc, seems a crucial part of a process where we gradually scale up challenges for agents that are capable of completing not just one challenge, but multiple ones.

PCG via machine learning could be a potentially promising approach to tackle this challenge. For example, Fan et al. [73] very recently showed that a neural network can learn from crowd-sourced elements such as descriptions of locations and characters to create multiplayer text adventure games. This idea of leveraging and integrating real world data to create games (also known as *Data Games*), was first proposed by Gustafsson et al. [74] and later extended to procedurally generate simple adventures games using open data from Wikipedia [75]. Another example of how to leverage advances in machine learning for PCG is the recent AI Dungeon 2 text adventure game [76]. In this game, players can type in any command and the system can respond to it reasonably well, creating the first never-ending text adventure. The system is built on OpenAI's GPT-2 language model [77], which was further fine-tuned on a number of text adventure stories. This work also highlights that machine learning techniques combined with PCG might lead to completely new types of games that would not have been possible without advanced AI methods.

6.3 Lifelong generation for lifelong learning

The problem of Lifelong Learning is that of continuously adapting and improving skills over a long lifetime of an agent, comprising many individual episodes, though not necessarily divided into episodes as currently thought of [78, 79]. This would require for an agent to build on previously learned skills as it faces increasingly harder or more complex, or just more varied, challenges. Lifelong learning is a problem, or maybe rather a setting, whose popularity has seemingly waxed and waned (under different names) as subsequent generations of researchers have discovered this challenge and then understood how hard it is. Within the artificial life community, the challenge of simulating *open-ended evolution* is closely related to that of lifelong learning. The idea behind open-ended evolution is to try to computationally replicate the process that allows nature to endlessly produce a diversity of interesting and complex artifacts. Environments such as Tierra [80] and Avida [81] were early attempts at realizing that possibility.

The procedural generation of environments and challenges is a great opportunity for lifelong learning, and might even be a precondition for lifelong learning to be practically possible. It is possible that earlier attempts at realizing lifelong learning have had limited success partly because the environments lacked sufficient challenges of the right complexity. The POET system shows one way of co-creating environments with agents [29]. However, there is a great outstanding research challenge in devising mechanisms for gradually growing or complexifying environments (see next Section) so as to generate the right problems at the right time for agents to continually learn.

6.4 New PCG-based RL benchmarks

A variety of benchmarks have been proposed to test the generalization abilities of RL algorithms. Justesen et al. [4] used procedurally generated levels in the General Video Game AI (GVG-AI) framework [82], to study overfitting of RL algorithms to different level distribution. In a similar vein to the work by Justesen et al. [4], levels in the CoinRun platform game are procedurally generated to quantify the ability of RL algorithms to generalize to never-before-seen levels [68, 83]. Another procedurally generated environment is the Unity game engine-based Obstacle Tower environment [84], which requires increasingly complex skills such as locomotion, planning, and puzzle-solving. Others have recently combined the Unity environment with GVGAI, creating UnityVGDL [85], which allows ML agents in Unity to be tested on a large selection of games.

Other setups that do not use PCG include the work by Nichol et al. [86], in which *Sonic the Hedgehog*^{TM} levels were separated into a training and test set to investigate how well RL algorithms generalize. In the Psychlab environment [87], agents are tested on known tasks from cognitive psychology, such as visual search or object tracking, making the results from simulated agents directly comparable to human results.

We propose the creation of PCG-based benchmarks in which the agent's environment and reward is non-stationary and becomes more and more complex over time. A starting point could be PCG approaches that are able to evolve the actual rules of a game (see section 6.2). New rules could be introduced based on agents' performance and estimates of their learning capacity.

Adaptation within trials is as important as adaptation between trials: a generator could generate increasingly difficult games, which are different enough in each trial that a policy that would not adapt within a trial would fail. The Animal-AI Environment [88], in which agents have to adapt to unforeseen challenges based on classical tests from animal cognition studies, shares similar ideas with the benchmarks we are proposing here but does not focus on procedurally generated environments and tasks.

6.5 From simulation to the real world

Procedurally generated environments have shown their potential in training robot policies that can cross the reality gap. Promising work includes approaches that try to learn the optimal parameters of a simulator, so that policies trained in that simulator work well with real data

[89, 90]. However, current approaches are still limited to lab settings, and we are far from being able to train robots that can deal with the messiness and diversity of tasks and environments encountered in the real world.

An intriguing opportunity is to train policies in much more diverse simulated environments than have been explored so far, with the hope that they will be able to cope better with a wider range of tasks when transferred to real physical environments. Both the Unity Simulation environment and Facebook's AI Habitat are taking a step in this direction. With Unity Simulation, Unity is aiming for simulation environments to work at scale, allowing developers to built digital twins of factories, warehouses or driving environments. Facebook's AI Habitat is designed to train embodied agents and robots in photo-realistic 3D environments to ultimately allow them to work in the real world.

In addition to developing more sophisticated machine learning models, one important research challenge in crossing the reality gap is the *content gap* [90]. Because the synthetic content that the agents are trained on typically only represents a limited set of scenarios that might be encountered in the real world, the agents will likely fail if they encounter situations that are too different from what they have seen before.

How to create PCG approaches that can limit this content gap and create large and diverse training environments, which prepare agents well for the real world tasks to come, is an important open research direction.

Acknowledgements

We would like to thank all the members of modl.ai, especially Niels Justesen, for comments on earlier drafts of this manuscript. We would also like to thank Andrzej Wojcicki, Rodrigo Canaan, Nataniel Ruiz, and the anonymous reviewers for additional comments and suggestions. Both authors (SR and JT) contributed equally to the conceptualization and writing of the paper.

References

[1] Adam Summerville, Sam Snodgrass, Matthew Guzdial, Christoffer Holmgård, Amy K Hoover, Aaron Isaksen, Andy Nealen, and Julian Togelius. Procedural content generation via machine learning (pcgml). *IEEE Transactions on Games*, 10(3):257–270, 2018.

[2] Moritz Hardt, Eric Price, Nati Srebro, et al. Equality of opportunity in supervised learning. In *Advances in neural information processing systems*, pages 3315–3323, 2016.

[3] Junhong Lin, Raffaello Camoriano, and Lorenzo Rosasco. Generalization properties and implicit regularization for multiple passes sgm. In *International Conference on Machine Learning*, pages 2340–2348, 2016.

[4] Niels Justesen, Ruben Rodriguez Torrado, Philip Bontrager, Ahmed Khalifa, Julian Togelius, and Sebastian Risi. Illuminating generalization in deep reinforcement learning through procedural level generation. *NeurIPS 2018 Workshop on Deep Reinforcement Learning*, 2018.

[5] Chiyuan Zhang, Oriol Vinyals, Remi Munos, and Samy Bengio. A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*, 2018.

[6] Avraham Ruderman, Richard Everett, Bristy Sikder, Hubert Soyer, Jonathan Uesato, Ananya Kumar, Charlie Beattie, and Pushmeet Kohli. Uncovering surprising behaviors in reinforcement learning via worst-case analysis. 2018.

[7] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[9] Patrice Y Simard, David Steinkraus, John C Platt, et al. Best practices for convolutional neural networks applied to visual document analysis. In *Icdar*, volume 3, 2003.

[10] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 23–30. IEEE, 2017.

[11] Max Jaderberg, Wojciech M Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castaneda, Charles Beattie, Neil C Rabinowitz, Ari S Morcos, Avraham Ruderman, et al. Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science*, 364(6443):859–865, 2019.

[12] Ruben Rodriguez Torrado, Ahmed Khalifa, Michael Cerny Green, Niels Justesen, Sebastian Risi, and Julian Togelius. Bootstrapping conditional gans for video game level generation. *arXiv preprint arXiv:1910.01603*, 2019.

[13] OpenAI, Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, Jonas Schneider, Nikolas Tezak, Jerry Tworek, Peter Welinder, Lilian Weng, Qiming Yuan, Wojciech Zaremba, and Lei Zhang. Solving rubik's cube with a robot hand, 2019.

[14] Raph Koster. *Theory of fun for game design.* " O'Reilly Media, Inc.", 2005.

[15] Adam James Summerville and Michael Mateas. Mystical tutor: A magic: The gathering design assistant via denoising sequence-to-sequence learning. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016.

[16] Adam Summerville and Michael Mateas. Super mario as a string: Platformer level generation via lstms. In *Proceedings of the DiGRA/FDG Joint Conference*, 2016.

[17] Sam Snodgrass and Santiago Ontanón. Controllable procedural content generation via constrained multi-dimensional markov chain sampling. In *IJCAI*, pages 780–786, 2016.

[18] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation: a taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3:172–186, 2011.

[19] Steve Dahlskog and Julian Togelius. A multi-level level generator. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

[20] Cameron Browne and Frederic Maire. Evolutionary game design. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(1):1–16, 2010.

[21] Julian Togelius, Mike Preuss, Nicola Beume, Simon Wessing, Johan Hagelbäck, Georgios N Yannakakis, and Corrado Grappiolo. Controllable procedural map generation via multiobjective evolution. *Genetic Programming and Evolvable Machines*, 14(2):245–277, 2013.

[22] Julian Togelius, Renzo De Nardi, and Simon M Lucas. Towards automatic personalised content creation for racing games. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 252–259. IEEE, 2007.

[23] Vanessa Volz, Jacob Schrum, Jialin Liu, Simon M Lucas, Adam Smith, and Sebastian Risi. Evolving mario levels in the latent space of a deep convolutional generative adversarial network. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 221–228. ACM, 2018.

[24] Philip Bontrager, Aditi Roy, Julian Togelius, Nasir Memon, and Arun Ross. Deepmasterprints: Generating masterprints for dictionary attacks via latent variable evolution. In *2018 IEEE 9th International Conference on Biometrics Theory, Applications and Systems (BTAS)*, pages 1–9. IEEE, 2018.

[25] Ahmed Khalifa, Philip Bontrager, Sam Earle, and Julian Togelius. Pcgrl: Procedural content generation via reinforcement learning. *arXiv preprint arXiv:2001.09212*, 2020.

[26] Philip Bontrager and Julian Togelius. Fully differentiable procedural content generation through generative playing networks. *arXiv preprint arXiv:2002.05259*, 2020.

[27] Anh Nguyen, Alexey Dosovitskiy, Jason Yosinski, Thomas Brox, and Jeff Clune. Synthesizing the preferred inputs for neurons in neural networks via deep generator networks. In *Advances in neural information processing systems*, pages 3387–3395, 2016.

[28] Ekin D Cubuk, Barret Zoph, Dandelion Mane, Vijay Vasudevan, and Quoc V Le. Autoaugment: Learning augmentation policies from data. *arXiv preprint arXiv:1805.09501*, 2018.

[29] Rui Wang, Joel Lehman, Jeff Clune, and Kenneth O Stanley. Paired open-ended trailblazer (poet): Endlessly generating increasingly complex and diverse learning environments and their solutions. *arXiv preprint arXiv:1901.01753*, 2019.

[30] Jonathan C Brant and Kenneth O Stanley. Minimal criterion coevolution: a new approach to open-ended search. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 67–74. ACM, 2017.

[31] Gillian Smith. An analog history of procedural content generation. In *FDG*, 2015.

[32] Ken Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985.

[33] Przemyslaw Prusinkiewicz. Graphical applications of l-systems. In *Proceedings of graphics interface*, volume 86, pages 247–253, 1986.

[34] John Von Neumann et al. The general and logical theory of automata. *1951*, pages 1–41, 1951.

[35] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*, 2017.

[36] Xiaodong Cui, Vaibhava Goel, and Brian Kingsbury. Data augmentation for deep neural network acoustic modeling. *IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP)*, 23(9):1469–1477, 2015.

[37] Robert Geirhos, Patricia Rubisch, Claudio Michaelis, Matthias Bethge, Felix A Wichmann, and Wieland Brendel. Imagenet-trained cnns are biased towards texture; increasing shape bias improves accuracy and robustness. *arXiv preprint arXiv:1811.12231*, 2018.

[38] Lilian Weng. Domain randomization for sim2real transfer. *lilianweng.github.io/lil-log*, 2019.

[39] Josh Tobin. Beyond domain randomization. sim2real.github.io, 2019.

[40] Fereshteh Sadeghi and Sergey Levine. Cad2rl: Real single-image flight without a single real image. *arXiv preprint arXiv:1611.04201*, 2016.

[41] Jonathan Tremblay, Aayush Prakash, David Acuna, Mark Brophy, Varun Jampani, Cem Anil, Thang To, Eric Cameracci, Shaad Boochoon, and Stan Birchfield. Training deep networks with synthetic data: Bridging the reality gap by domain randomization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 969–977, 2018.

[42] Aayush Prakash, Shaad Boochoon, Mark Brophy, David Acuna, Eric Cameracci, Gavriel State, Omer Shapira, and Stan Birchfield. Structured domain randomization: Bridging the reality gap by context-aware synthetic data. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 7249–7255. IEEE, 2019.

[43] Wenhao Yu, C Karen Liu, and Greg Turk. Policy transfer with strategy optimization. *arXiv preprint arXiv:1810.05751*, 2018.

[44] Sergey Zakharov, Wadim Kehl, and Slobodan Ilic. Deceptionnet: Network-driven domain randomization. *arXiv preprint arXiv:1904.02750*, 2019.

[45] Jürgen Schmidhuber. Curious model-building control systems. In *Proc. international joint conference on neural networks*, pages 1458–1463, 1991.

[46] Julian Togelius and Jürgen Schmidhuber. An experiment in automatic game design. In *Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On*, pages 111–118. IEEE, 2008.

[47] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[48] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[49] Michael Cook, Simon Colton, and Jeremy Gow. The angelina videogame design system—part i. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(2):192–203, 2016.

[50] Daniel Karavolos, Antonios Liapis, and Georgios N Yannakakis. A multi-faceted surrogate model for search-based procedural content generation. *IEEE Transactions on Games*, 2019.

[51] S. Risi, J. Lehman, D B D'Ambrosio, R. Hall, and Kenneth O Stanley. Petalz: Search-based procedural content generation for the casual gamer. *Computational Intelligence and AI in Games, IEEE Transactions on*, PP(99):1–1, 2015.

[52] Michael Cook and Simon Colton. Multi-faceted evolution of simple arcade games. In *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pages 289–296. IEEE, 2011.

[53] Thorbjørn S Nielsen, Gabriella AB Barros, Julian Togelius, and Mark J Nelson. Towards generating arcade game rules with vgdl. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 185–192. IEEE, 2015.

[54] William Cachia, Antonios Liapis, and Georgios N Yannakakis. Multi-level evolution of shooter levels. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 2015.

[55] Laura Calle, Juan J Merelo, Antonio Mora-García, and José-Mario García-Valdez. Free form evolution for angry birds level generation. In *International Conference on the Applications of Evolutionary Computation (Part of EvoStar)*, pages 125–140. Springer, 2019.

[56] Erin Jonathan Hastings, Ratan K Guha, and Kenneth O Stanley. Automatic content generation in the galactic arms race video game. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(4):245–263, 2009.

[57] Alex Pantaleev. In search of patterns: Disrupting rpg classes through procedural content generation. In *Proceedings of the The third workshop on Procedural Content Generation in Games*, page 4. ACM, 2012.

[58] Daniele Gravina, Ahmed Khalifa, Antonios Liapis, Julian Togelius, and Georgios N Yannakakis. Procedural content generation through quality diversity. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2019.

[59] Antoine Cully, Jeff Clune, Danesh Tarapore, and Jean-Baptiste Mouret. Robots that can adapt like animals. *Nature*, 521(7553):503, 2015.

[60] Ahmed Khalifa, Michael Cerny Green, Gabriella Barros, and Julian Togelius. Intentional computational level design. In *Genetic and Evolutionary Computation Conference (GECCO)*, 2019.

[61] Michael Mateas Gillian Smith, Jim Whitehead. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *IEEE Transactions on Computational Intelligence and AI in Games (TCIAIG), Special Issue on Procedural Content Generation*, 3, 2011.

[62] Adam M Smith and Michael Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):187–200, 2011.

[63] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[64] Oleg Klimov. Carracing-v0. https://gym.openai.com/envs/CarRacing-v0/, 2016.

[65] David Ha and Jürgen Schmidhuber. Recurrent world models facilitate policy evolution. In *Advances in Neural Information Processing Systems*, pages 2450–2462, 2018.

[66] David Ha. Evolving stable strategies. http://blog.otoro.net/2017/11/12/evolving-stable-strategies/, 2017.

[67] Thomas Lukasiewicz, Yuhang Song, Lianlong Wu, and Zhenghua Xu. Arena: A general evaluation platform and building toolkit for multi- agent intelligence. In *Proceedings of the 34th National Conference on Artificial Intelligence*, 2020.

[68] Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying generalization in reinforcement learning. *arXiv preprint arXiv:1812.02341*, 2018.

[69] Jeff Clune. AI-GAs: AI-generating algorithms, an alternate paradigm for producing general artificial intelligence. *arXiv preprint arXiv:1905.10985*, 2019.

[70] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[71] Mark J Nelson and Michael Mateas. Towards automated game design. In *Congress of the Italian Association for Artificial Intelligence*, pages 626–637. Springer, 2007.

[72] Jose M Font, Tobias Mahlmann, Daniel Manrique, and Julian Togelius. A card game description language. In *European Conference on the Applications of Evolutionary Computation*, pages 254–263. Springer, 2013.

[73] Angela Fan, Jack Urbanek, Pratik Ringshia, Emily Dinan, Emma Qian, Siddharth Karamcheti, Shrimai Prabhumoye, Douwe Kiela, Tim Rocktaschel, Arthur Szlam, and Jason Weston. Generating interactive worlds with text, 2019.

[74] Marie Gustafsson Friberger, Julian Togelius, Andrew Borg Cardona, Michele Ermacora, Anders Mousten, Martin Møller Jensen, Virgil-Alexandu Tanase, and Ulrik Brøndsted. Data games. In *Proceedings of the The Fourth workshop on Procedural Content Generation in Games;*. ACM, 2013.

[75] Gabriella AB Barros, Antonios Liapis, and Julian Togelius. Playing with data: Procedural generation of adventures from open data. In *DiGRA/FDG*, 2016.

[76] Nick Walton. AI dungeon 2. ttps://aidungeon.io/, 2019.

[77] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019.

[78] Sebastian Thrun and Tom M Mitchell. Lifelong robot learning. *Robotics and autonomous systems*, 15(1-2):25–46, 1995.

[79] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural Networks*, 2019.

[80] Thomas S Ray. An evolutionary approach to synthetic biology: Zen and the art of creating life. *Artificial Life*, 1(1_2):179–209, 1993.

[81] Chris Adami, C Titus Brown, and W Kellogg. Evolutionary learning in the 2d artificial life system 'avida'. In *Artificial life IV*, volume 1194, pages 377–381. MIT press Cambridge, MA, 1994.

[82] Diego Perez-Liebana, Jialin Liu, Ahmed Khalifa, Raluca D Gaina, Julian Togelius, and Simon M Lucas. General video game ai: a multi-track framework for evaluating agents, games and content generation algorithms. *arXiv preprint arXiv:1802.10363*, 2018.

[83] Karl Cobbe, Christopher Hesse, Jacob Hilton, and John Schulman. Leveraging procedural generation to benchmark reinforcement learning. *arXiv preprint arXiv:1912.01588*, 2019.

[84] Arthur Juliani, Ahmed Khalifa, Vincent-Pierre Berges, Jonathan Harper, Hunter Henry, Adam Crespi, Julian Togelius, and Danny Lange. Obstacle tower: A generalization challenge in vision, control, and planning. *arXiv preprint arXiv:1902.01378*, 2019.

[85] Mads Johansen, Martin Pichlmair, and Sebastian Risi. Video game description language environment for unity machine learning agents. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2019.

[86] Alex Nichol, Vicki Pfau, Christopher Hesse, Oleg Klimov, and John Schulman. Gotta learn fast: A new benchmark for generalization in RL. *arXiv preprint arXiv:1804.03720*, 2018.

[87] Joel Z Leibo, Cyprien de Masson d'Autume, Daniel Zoran, David Amos, Charles Beattie, Keith Anderson, Antonio García Castañeda, Manuel Sanchez, Simon Green, Audrunas Gruslys, et al. Psychlab: a psychology laboratory for deep reinforcement learning agents. *arXiv preprint arXiv:1801.08116*, 2018.

[88] Benjamin Beyret, José Hernández-Orallo, Lucy Cheke, Marta Halina, Murray Shanahan, and Matthew Crosby. The animal-AI environment: Training and testing animal-like artificial cognition. *arXiv preprint arXiv:1909.07483*, 2019.

[89] Nataniel Ruiz, Samuel Schulter, and Manmohan Chandraker. Learning to simulate. *arXiv preprint arXiv:1810.02513*, 2018.

[90] Amlan Kar, Aayush Prakash, Ming-Yu Liu, Eric Cameracci, Justin Yuan, Matt Rusiniak, David Acuna, Antonio Torralba, and Sanja Fidler. Meta-sim: Learning to generate synthetic datasets. *arXiv preprint arXiv:1904.11621*, 2019.